UNIT V

BRANCH BOUND AND RANDOMIZED ALGORITHM

Branch and bound: 0/1 Knapsack - Travelling Salesman Problem. Randomized algorithm: Hiring Problem - Matrix Chain Multiplication - Randomized Quick Sort. Introduction to PN problems - Introduction to NP problems - NP Complete.

INTRODUCTION

General method:

The design technique known as ***branch and bound*** is very similar to backtracking (seen in unit 4) in that it searches a tree model of the solution space and is applicable to a wide variety of discrete combinatorial problems.

Each node in the combinatorial tree generated in the last Unit defines a *problem state*. All paths from the root to other nodes define the ***state space*** of the problem.

Solution states are those problem states *'s'* for which the path from the root to *'s'* defines a tuple in the solution space. The leaf nodes in the combinatorial tree are the solution states.

Answer states are those solution states *'s'* for which the path from the root to *'s'* defines a tuple that is a member of the set of solutions (i.e,it satisfies the implicit constraints) of the problem.

The tree organization of the solution space is referred to as the ***state space tree*.**

A node which has been generated and all of whose children have not yet been generated is called a***live node***

The ***live node*** whose children are currently being generated is called the *E*-node (node being expanded).

A ***dead node***is a generated node, which is not to be expanded further or all of whose children have been generated.

Bounding functions are used to kill live nodes without generating all their children.

Depth first node generation with bounding function is called backtracking. State generation methods in which the *E*-node remains the *E*-node until it is dead lead to ***branch-and-bound method*.**

The term branch-and-bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node.

In branch-and-bound terminology breadth first search(BFS)- like state space search will be called FIFO (First In First Output) search as the list of live nodes is a first -in-first -out list(or queue).

A ***D-search*** (depth search) state space search will be called LIFO (Last In First Out) search, as the list of live nodes is a last-in-first-out list (or stack).

Bounding functions are used to help avoid the generation of sub trees that do not contain an answer node.

The branch-and-bound algorithms search a tree model of the solution space to get the solution. However, this type of algorithms is oriented more toward optimization. An algorithm of this type specifies a real -valued cost function for each of the nodes that appear in the search tree.

Usually, the goal here is to find a configuration for which the cost function is minimized. The branch-and-bound algorithms are rarely simple. They tend to be quite complicated in many cases.

0/1 KNAPSACK PROBLEM

Consider the instance: M = 15, n = 4, (P1, P2, P3, P4) = (10, 10, 12, 18) and (w1, w2, w3, w4) = (2, 4, 6, 9).

0/1 knapsack problem can be solved by using branch and bound technique. In this problem we will calculate lower bound and upper bound for each node.

Place first item in knapsack. Remaining weight of knapsack is 15 – 2 = 13. Place next item w2 in knapsack and the remaining weight of knapsack is 13 – 4 = 9. Place next item w3 in knapsack then the remaining weight of knapsack is 9 – 6 = 3. No fractions are allowed in calculation of upper bound so w4 cannot be placed in knapsack.

Profit = P1 + P2 + P3 = 10 + 10 + 12

So, Upper bound = 32

To calculate lower bound we can place w4 in knapsack since fractions are allowed in calculation of lower bound.

Lower bound = $10 + 10 + 12 + (\frac{3}{9} \times 18) = 32 + 6 = 38$

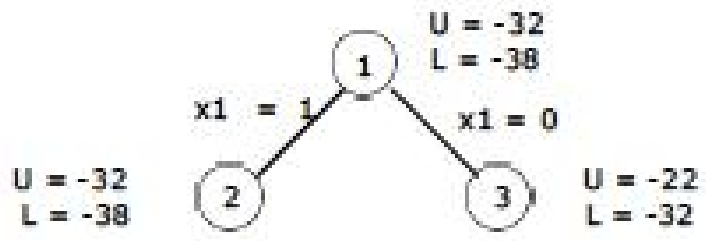
Knapsack problem is maximization problem but branch and bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

Therefore,

Upper bound (U) = -32

Lower bound (L) = -38

We choose the path, which has minimum difference of upper bound and lower bound. If the difference is equal, then we choose the path by comparing upper bounds and we discard node with maximum upper bound.



Now we will calculate upper bound and lower bound for nodes 2, 3. For node 2, x1= 1, means we should place first item in the knapsack.

$$U = 10 + 10 + 12 = 32, make\ it\ as\ -32$$

$$L = 10 + 10 + 12 + \frac{3}{9} \times 18 = 32 + 6 = 38, make\ it\ as\ -38$$

For node 3, x1 = 0, means we should not place first item in the knapsack.

$$U = 10 + 12 = 22, make\ it\ as\ -22$$

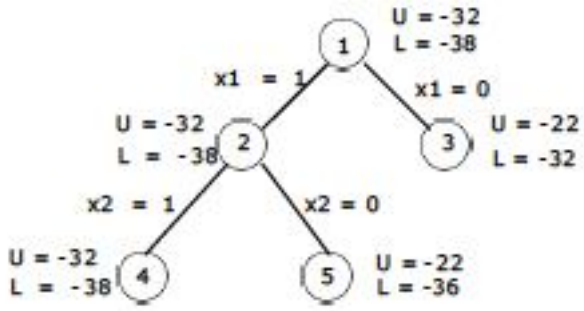
$$L = 10 + 12 + \frac{5}{9} \times 18 = 10 + 12 + 10 = 32, make\ it\ as\ -32$$

Next, we will calculate difference of upper bound and lower bound for nodes 2, 3

For node 2, U – L = -32 + 38 = 6

For node 3, U – L = -22 + 32 = 10

Choose node 2, since it has minimum difference value of 6.

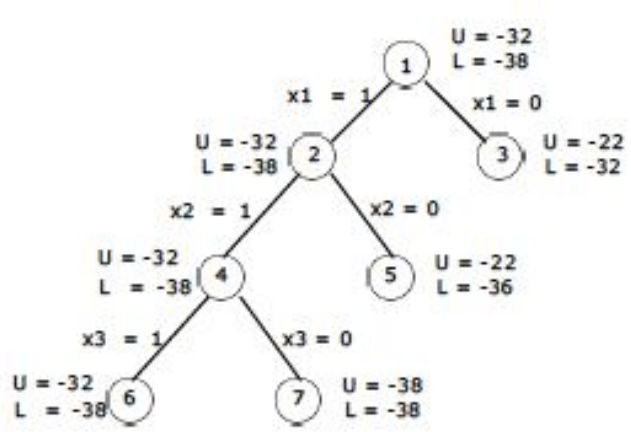


Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 4, U – L = -32 + 38 = 6

For node 5, U – L = -22 + 36 = 14

Choose node 4, since it has minimum difference value of 6.

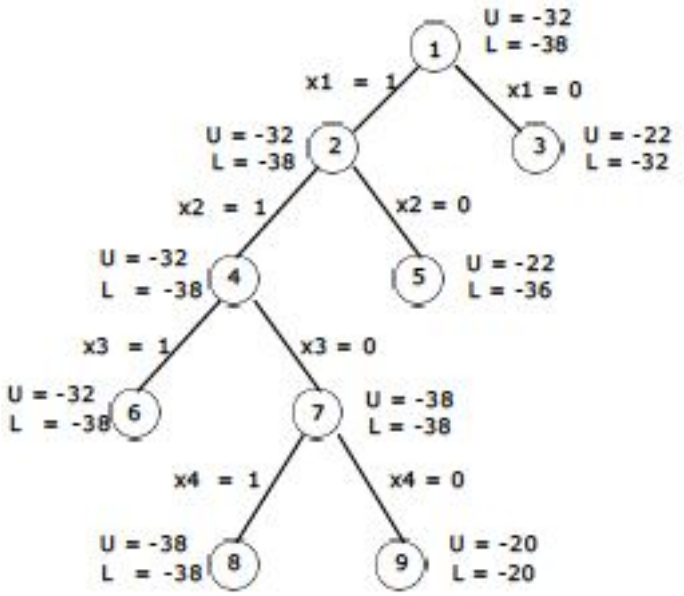


Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9.

For node 6, U – L = -32 + 38 = 6

For node 7, U – L = -38 + 38 = 0

Choose node 7, since it is minimum difference value of 0.



Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9.

For node 8, U – L = -38 + 38 = 0

For node 9, U – L = -20 + 20 = 0

Here the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.

Consider the path from 1 →2→4→7→8

$X_1 = 1$

$X_2 = 1$

$X_3 = 0$

X4 = 1

The solution for 0/1 Knapsack problem is (x1, x2, x3, x4) = (1, 1, 0, 1) Maximum profit is:

$$\sum Pi\, xi = 10 \times 1 + 10 \times 1 + 12 \times 0 + 18 \times 1$$
$$= 10 + 10 + 18 = 38.$$

TRAVELLING SALESMAN PROBLEM

It is algorithmic procedures similar to backtracking in which a new branch is chosen and is there (bound there) until new branch is choosing for advancing. This technique is implemented in the traveling salesman problem [TSP] which are asymmetric (Cij <>Cij) where this technique is an effective procedure.

Steps involved in this procedure are as follows:

STEP 0: Generate cost matrix C [for the given graph g]

STEP 1: [*ROW REDUCTION]* For all rows do step 2

STEP 2: Find least cost in a row and negate it with rest of the elements

STEP 3: [COLUMN REDUCTION] Use cost matrix- Row reduced one for all columns do STEP 4.

STEP 4: Find least cost in a column and negate it with rest of the elements

STEP 5: Preserve cost matrix C [which row reduced first and then column reduced] for the i^{th} time.

STEP 6: Enlist all edges (i, j) having cost = 0.

STEP 7: Calculate effective cost of the edges. $\sum$ (i, j)=least cost in the i^{th} row excluding (i, j) + least cost in the j^{th} column excluding (i, j).

STEP 8: Compare all effective cost and pick up the largest l. If two or more have same cost then arbitrarily choose any one among them

STEP 9: Delete (i, j) means delete i^{th} row and j^{th} column change (j, i) value to infinity. (Used to avoid infinite loop formation) If (i,j) not present, leave it.

STEP 10: Repeat step 1 to step 9 until the resultant cost matrix having order of 2*2 and reduce it. (Both R.R and C.C)

STEP 11: Use preserved cost matrix Cn, Cn-1… C1. Choose an edge [i, j] having value =0, at the first time for a preserved matrix and leave that matrix.

STEP 12: Use result obtained in Step 11 to generate a complete tour.

EXAMPLE: *Given graph G*

MATRIX:

	1	2	3	4	5
1	α	25	40	31	27
2	5	α	17	30	25
3	19	15	α	6	1
4	9	50	24	α	6
5	22	8	7	10	α

PHASE I

STEP 1: Row Reduction C

C1 [ROW REDUCTION:

	1	2	3	4	5
1	α	0	15	6	2
2	0	α	12	25	20
3	18	14	α	5	0
4	3	44	18	α	0
5	15	1	0	3	α

STEP 3:
C1 [Column Reduction]

	1	2	3	4	5
1	α	0	15	3	2
2	0	α	12	22	20
3	18	14	α	2	0
4	3	44	18	α	0
5	15	1	0	0	α

STEP 5:

Preserve the above in C1,

	1	2	3	4	5
1	α	0	15	3	2
2	0	α	12	22	20
3	18	14	α	2	0
4	3	44	18	α	0
5	15	1	0	0	α

STEP 6:

L= $\{(1,2),(2,1),(3,5),(4,5),(5,3),(5,4)\}$

STEP 7:

Calculation of effective cost [E.C]
(1,2) = 2+1 =3
(2,1) = 12+3 = 15
(3,5) = 2+0 =2
(4,5) = 3+0 = 3
(5,3) = 0+12 = 12
(5,4) = 0+2 = 2

STEP 8:

L having edge (2,1) is the largest.

STEP 9: Delete (2,1) from C1 and make change in it as (1,2) →α if exists.

Now Cost Matrix =

	2	3	4	5
1	α	15	3	2
3	14	α	2	0
4	44	18	α	0
5	1	0	0	α

STEP 10: The Cost matrix ≠ 2 x 2.
Therefore, go to step 1.

PHASE II:

STEP1: C2(R, R)

	2	3	4	5
1	α	13	1	0
3	14	α	2	0
4	44	18	α	0
5	1	0	0	α

STEP 3: C2 (C, R)

	2	3	4	5
1	α	13	1	0
3	13	α	2	0
4	43	18	α	0
5	0	0	0	α

STEP 5: Preserve the above in C2

C2 =

	2	3	4	5
1	α	13	1	0
3	13	α	2	0
4	43	18	α	0
5	0	0	0	α

STEP 6:

L= {(1,5), (3,5), (4,5), (5,2), (5,3), (5,4)}

STEP 7: calculation of E.C.

(1,5) = 1+0 =1
(3,5) = 2+0 =2
(4,5) = 18+0 =18
(5,2) = 0+13 =13
(5,3) = 0+13 =13
(5,4) = 0+1 =1

STEP 8: L having an edge (4,5) is the largest.

STEP 9: Delete (4,5) from C2 and make change in it as (5,4) = α if exists.

Now, cost matrix

	2	3	4
1	α	13	1
3	13	α	2
5	0	0	α

STEP 10: THE cost matrix ≠ 2x2 hence go to step 1

PHASE III:

STEP 1: C3 (R, R)

	2	3	4
1	α	12	0
3	11	α	0
5	0	0	α

STEP 3: C3 (C, R)

	2	3	4
1	α	12	0
3	11	α	0
5	0	0	α

STEP 5: preserve the above in C3

STEP 6: L={(1,4), (3,4), (5,2), (5,3)}

STEP 7: calculation of E.C

(1,4)=12+0=12
(3,4)=11+0=11
(5,2)=0+11=11
(5,3)=0+12=12

STEP 8: Here we are having two edges (1,4) and (5,3) with cost = 12. Hence arbitrarily choose (1,4)

STEP 9: Delete (i,j) →(1,4) and make change in it (4,1) = α if exists.

Now cost matrix is

	2	3
3	11	α
5	0	0

STEP 10: We have got 2x2 matrix

C4 (RR)=

	2	3
3	0	α
5	0	0

C4 (C, R) =

	2	3
3	0	α
5	0	0

Therefore,C4 =

	2	3
3	0	α
5	0	0

STEP 11: LIST C1, C2, C3 AND C4

C4

	2	3
3	0	α
5	0	0

C3

	2	3	4
1	α	12	0
3	11	α	0
5	0	0	α

C2 =

	2	3	4	5
1	α	13	1	0
3	13	α	2	0
4	43	18	α	0
5	0	0	0	α

C1 =

	1	2	3	4	5
1	α	0	15	3	2
2	0	α	12	22	20
3	18	14	α	2	0
4	3	44	18	α	0
5	15	1	0	0	α

STEP 12:

i) Use C4 =

	2	3
3	0	α
5	0	0

Pick up an edge (I, j) =0 having least index

Here (3,2) =0

Hence, T← (3,2)

Use C3 =

	2	3	4
1	α	12	0
3	11	α	0
5	0	0	α

Pick up an edge (i, j) =0 having least index

Here (1,4) =0

Hence, T←(3,2), (1,4)

Use C2=

	2	3	4	5
1	α	13	1	0
3	13	α	2	0
4	43	18	α	0
5	0	0	0	α

Pick up an edge (i, j) with least cost index.

Here (1,5) → not possible because already chosen index i (i=j)
(3,5) → not possible as already chosen index.
(4,5) →0

Hence, T← (3,2), (1,4), (4,5)

Use C1 =

	1	2	3	4	5
1	α	0	15	3	2
2	0	α	12	22	20
3	18	14	α	2	0
4	3	44	18	α	0
5	15	1	0	0	α

Pick up an edge (i, j) with least index

(1,2) → Not possible
(2,1) → Choose it

HENCE T← (3,2), (1,4), (4,5), (2,1)

SOLUTION:

From the above list
3—2—1—4—5

This result now, we have to return to the same city where we started (Here 3).

Final result:

3—2—1—4—5—3

Cost is 15+15+31+6+7=64

Traveling Sale Person Problem:

By using dynamic programming algorithm we can solve the problem with time complexity of $O(n^2 2^n)$ for worst case. This can be solved by branch and bound technique using efficient bounding function. The time complexity of traveling sale person problem using LC branch and bound is $O(n^2 2^n)$ which shows that there is no change or reduction of complexity than previous method.

We start at a particular node and visit all nodes exactly once and come back to initial node with minimum cost.

Let G = (V, E) is a connected graph. Let C(i, J) be the cost of edge <i, j>. $c_{ij} = \infty$ if <i, j> $\notin$E and let |V| = n, the number of vertices. Every tour starts at vertex 1 and ends at the same vertex. So, the solution space is given by S = {1, π, 1 | π is a

permutation of (2, 3, . . . , n)} and |S| = (n – 1)!. The size of S can be reduced by restricting S so that $(1, i_1, i_2, \ldots . i_{n-1}, 1) \in S$ iff $\langle i_j, i_{j+1} \rangle \in E$, $0 \leq j \leq n - 1$ and $i_0 = i_n = 1$.

Procedure for solving traveling sale person problem:

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. A row (column) is said to be reduced if it contain at least one zero and all-remaining entries are non-negative. This can be done as follows:

   a) *Row reduction:* Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.

   b) Find the sum of elements, which were subtracted from rows.

   c) Apply column reductions for the matrix obtained after row reduction.

      Column reduction: Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.

   d) Find the sum of elements, which were subtracted from columns.

   e) Obtain the cumulative sum of row wise reduction and column wise reduction.

      Cumulative reduced sum = Row wise reduction sum + column wise reduction sum.

      Associate the cumulative reduced sum to the starting state as lower bound and ∞ as upper bound.

2. Calculate the reduced cost matrix for every node R. Let A is the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge <i, j> in the tour. If S is not a leaf node, then the reduced cost matrix for S may be obtained as follows:

   a) Change all entries in row i and column j of A to ∞.

   b) Set A (j, 1) to ∞.

   c) Reduce all rows and columns in the resulting matrix except for rows and column containing only ∞. Let r is the total amount subtracted to reduce the matrix.

   c) Find $c(S) = c(R) + A(i, j) + r$, where 'r' is the total amount subtracted to reduce the matrix, $c(R)$ indicates the lower bound of the i^{th} node in (i, j) path and $c(S)$ is called the cost function.

3. Repeat step 2 until all nodes are visited.

Example:

Find the LC branch and bound solution for the traveling sale person problem whose cost matrix is as follows:

The cost matrix is $\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$

Step 1: Find the reduced cost matrix.

Apply row reduction method:

Deduct 10 (which is the minimum) from all values in the 1st row.
Deduct 2 (which is the minimum) from all values in the 2nd row.
Deduct 2 (which is the minimum) from all values in the 3rd row.
Deduct 3 (which is the minimum) from all values in the 4th row.
Deduct 4 (which is the minimum) from all values in the 5th row.

The resulting row wise reduced cost matrix = $\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 0 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$

Row wise reduction sum = 10 + 2 + 2 + 3 + 4 = 21

Now apply column reduction for the above matrix:

Deduct 1 (which is the minimum) from all values in the 1st column.
Deduct 3 (which is the minimum) from all values in the 3rd column.

The resulting column wise reduced cost matrix (A) = $\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$

Column wise reduction sum = 1 + 0 + 3 + 0 + 0 = 4

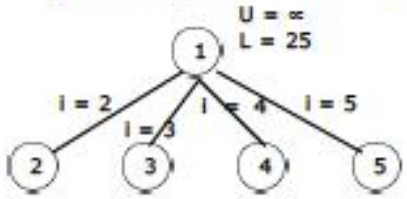
Cumulative reduced sum = row wise reduction + column wise reduction sum.
= 21 + 4 = 25.

This is the cost of a root i.e., node 1, because this is the initially reduced cost matrix.

The lower bound for node is 25 and upper bound is ∝.

Starting from node 1, we can next visit 2, 3, 4 and 5 vertices. So, consider to explore the paths (1, 2), (1, 3), (1, 4) and (1, 5).

The tree organization up to this point is as follows:



Variable 'i' indicates the next node to visit.

Step 2:

Consider the path (1, 2):

Change all entries of row 1 and column 2 of A to ∝ and also set A(2, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 0 + 0 + 0 + 0 = 0
Column reduction sum = 0 + 0 + 0 + 0 = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $c(S) = c(R) + A(1,2) + r$

$c(S) = 25 + 10 + 0 = 35$

Consider the path (1, 3):

Change all entries of row 1 and column 3 of A to ∝ and also set A(3, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 11
Cumulative reduction (r) = 0 + 11 = 11

Therefore, as $c(S) = c(R) + A(1,3) + r$

$c(S) = 25 + 17 + 11 = 53$

Consider the path (1, 4):

Change all entries of row 1 and column 4 of A to ∝ and also set A(4, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1,4) + r$

$\hat{c}(S) = 25 + 0 + 0 = 25$

Consider the path (1, 5):

Change all entries of row 1 and column 5 of A to ∝ and also set A(5, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

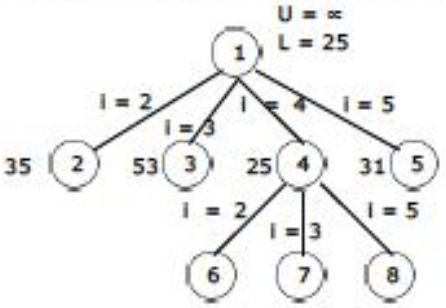
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 5
Column reduction sum = 0
Cumulative reduction (r) = 5 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1,5) + r$

$\hat{c}(S) = 25 + 1 + 5 = 31$

The tree organization up to this point is as follows:



The cost of the paths between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25 and (1, 5) = 31. The cost of the path between (1, 4) is minimum. Hence the matrix obtained for path (1, 4) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths are (4, 2), (4, 3) and (4, 5).

Consider the path (4, 2):

Change all entries of row 4 and column 2 of A to ∝ and also set A(2, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $c(S) = c(R) + A(4,2) + r$

$c(S) = 25 + 3 + 0 = 28$

Consider the path (4, 3):

Change all entries of row 4 and column 3 of A to ∝ and also set A(3, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1\infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty\infty & \infty & \infty \\ 00 & \infty & \infty & \infty \end{bmatrix}$$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(4,3) + r$

$$\hat{c}(S) = 25 + 12 + 13 = 50$$

Consider the path (4, 5):

Change all entries of row 4 and column 5 of A to ∝ and also set A(5, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is

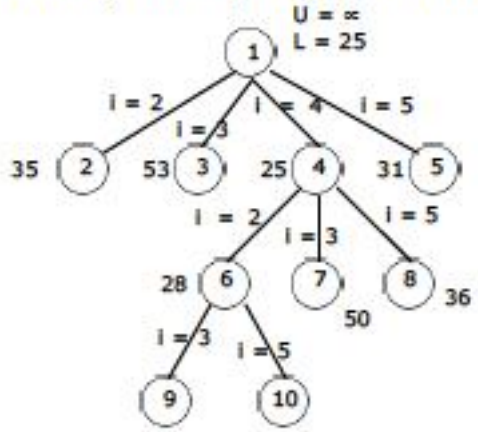
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1\infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty\infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

Row reduction sum = 11
Column reduction sum = 0
Cumulative reduction (r) = 11+0 = 11

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(4,5) + r$

$$\hat{c}(S) = 25 + 0 + 11 = 36$$

The tree organization up to this point is as follows:



The cost of the paths between (4, 2) = 28, (4, 3) = 50 and (4, 5) = 36. The cost of the path between (4, 2) is minimum. Hence the matrix obtained for path (4, 2) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths are (2, 3) and (2, 5).

Consider the path (2, 3):

Change all entries of row 2 and column 3 of A to ∝ and also set A(3, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(2,3) + r$

$\hat{c}(S) = 28 + 11 + 13 = 52$

Consider the path (2, 5):

Change all entries of row 2 and column 5 of A to ∝ and also set A(5, 1) to ∝.

$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

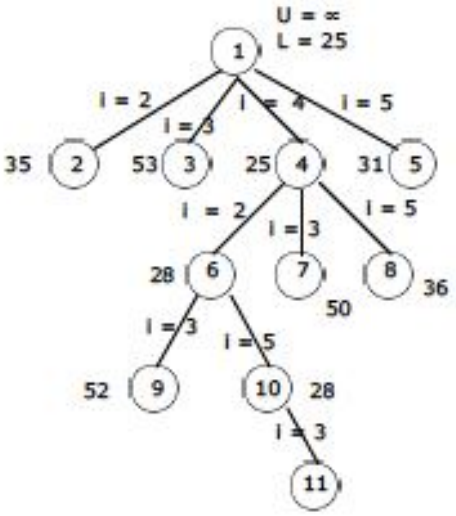
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(2,5) + r$

$\hat{c}(S) = 28 + 0 + 0 = 28$

The tree organization up to this point is as follows:



The cost of the paths between (2, 3) = 52 and (2, 5) = 28. The cost of the path between (2, 5) is minimum. Hence the matrix obtained for path (2, 5) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths is (5, 3).

Consider the path (5, 3):

Change all entries of row 5 and column 3 of A to ∝ and also set A(3, 1) to ∝. Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

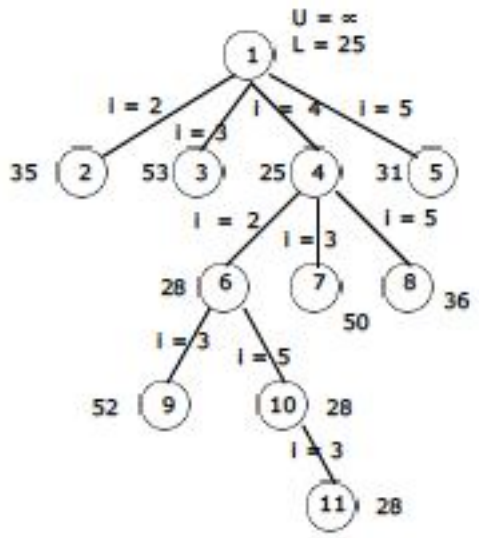
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $c(S) = c(R) + A(5, 3) + r$

$c(S) = 28 + 0 + 0 = 28$

The overall tree organization is as follows:



The path of traveling sale person problem is:

1 → 4 → 2 → 5 → 3 → 1

The minimum cost of the path is: 10 + 6 +2+ 7 + 3 = 28.

Difference between Backtracking and Branch and Bound

Back tracking	Branch and Bound
It is used to find all possible solutions available to the problem	It is used to solve optimization problem
It traverse tree by DFS(Depth First Search).	It may traverse the tree in any manner, DFS or BFS
It realizes that it has made a bad choice & undoes the last choice by backing up	It realizes that it already has a better optimal solution that the pre-solution leads to so it abandons that pre-solution.
It search the state space tree until it found a solution	It completely searches the state space tree to get optimal solution
It involves feasibility function	It involves bounding function
It is more often not applied to non-optimization.	It can be applied only to optimization problems.

RANDOMIZED ALGORITHMS

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm. For example, in Randomized Quick Sort, we use random number to pick the next pivot (or we randomly shuffle the array).

Classification of Randomized algorithms

Las Vegas: These algorithms always produce correct or optimum result. Time complexity of these algorithms is based on a random value and time complexity is evaluated as expected value. For example, Randomized QuickSort always sorts an input array and expected worst case time complexity of QuickSort is O(nLogn).

Monte Carlo: Produce correct or optimum result with some probability. These algorithms have deterministic running time and it is generally easier to find out worst case time complexity. For example implementation of Karger's Algorithm produces minimum cut with probability greater than or equal to $1/n^2$ (n is number of vertices) and has worst case time complexity as O(E). Another example is Fermet Method for Primality Testing.

Example to Understand Classification:

Consider a binary array where exactly half elements are 0 and half are 1. The task is to find index of any 1.

A Las Vegas algorithm for this task is to keep picking a random element until we find a 1. A Monte Carlo algorithm for the same is to keep picking a random element until we either find 1 or we have tried maximum allowed times say k.

The Las Vegas algorithm always finds an index of 1, but time complexity is determined as expect value. The expected number of trials before success is 2, therefore expected time complexity is O(1). The Monte Carlo Algorithm finds a 1 with probability $[1 – (1/2)^k]$. Time complexity of Monte Carlo is O(k) which is deterministic

Applications and Scope:

- Consider a tool that basically does sorting. Let the tool be used by many users and there are few users who always use tool for already sorted array. If the tool uses simple (not randomized) QuickSort, then those few users are always going to face worst case situation. On the other hand if the tool uses Randomized QuickSort, then there is no user that always gets worst case. Everybody gets expected O(n Log n) time.
- Randomized algorithms have huge applications in Cryptography.
- Load Balancing.

- Number-Theoretic Applications: Primality Testing
- Data Structures: Hashing, Sorting, Searching, Order Statistics and Computational Geometry.
- Algebraic identities: Polynomial and matrix identity verification. Interactive proof systems.
- Mathematical programming: Faster algorithms for linear programming, Rounding linear program solutions to integer program solutions
- Graph algorithms: Minimum spanning trees, shortest paths, minimum cuts.
- Counting and enumeration: Matrix permanent Counting combinatorial structures.
- Parallel and distributed computing: Deadlock avoidance distributed consensus.
- Probabilistic existence proofs: Show that a combinatorial object arises with non-zero probability among objects drawn from a suitable probability space.
- Derandomization: First devise a randomized algorithm then argue that it can be derandomized to yield a deterministic algorithm.

HIRING PROBLEM

We will now begin our investigation of randomized algorithms with a toy problem:

- You want to hire an office assistant from an employment agency.
- You want to interview candidates and determine if they are better than the current assistant and if so replace the current assistant.
- You are going to eventually interview every candidate from a pool of n candidates.
- You want to always have the best person for this job, so you will replace an assistant with a better one as soon as you are done the interview.
- However, there is a cost to fire and then hire someone.
- You want to know the expected price of following this strategy until all n candidates have been interviewed.

Hire-Assistant(n)

1 **best** ← **0** candidate 0 is a least-qualified dummy candidate

2 for ***i*** ← **1** to **n**

3 do interview candidate ***i in random permutation***

4 if candidate ***i*** is better than candidate ***best***

5 then **best** ← ***i***

6 hire candidate ***i***

Total Cost and Cost of Hiring

- Interviewing has a low cost $\boldsymbol{ci}$.
- Hiring has a high cost $\boldsymbol{c_h}$.
- Let **n** be the number of candidates to be interviewed andlet **m** be the number of people hired.
- The total cost then goes as $O(\boldsymbol{n} * \boldsymbol{ci} + \boldsymbol{m} * \boldsymbol{ch})$
- The number of candidates is fixed so the part of thealgorithm we want to focus on is the $\boldsymbol{m} * \boldsymbol{ch}$term.
- This term governs the cost of hiring.

Worst-case Analysis

- In the worst case, everyone we interview turns out to be better than the person we currently have.
- In this case, the hiring cost for the algorithm will be O(n*ch).
- This bad situation presumably doesn't typically happen so it is interesting to ask what happens in the average case.

Probabilistic analysis

- Probabilistic analysis is the use of probability to analyze problems.
- One important issue is what is the distribution of inputs to the problem.
- For instance , we could assume all orderings of candidates are equally likely.
- That is, we consider all functions rank: [0..n] --> [0..n] where rank[i] is supposed to be the **i**th candidate that we interview. So <rank(1), rank(2), …rank(n)> should be a permutation of of <1,…,n>
- There are n! many such permutations and we want each to be equally likely.
- If this is the case, the ranks form a uniform random permutation.

Randomized algorithms

- In order to use probabilistic analysis, we need to know something about the distribution of the inputs.
- Unfortunately, often little is known about this distribution.
- We can nevertheless use probability and analysis as a tool for algorithm design by having the algorithm we run do some kind of randomization of the inputs.
- This could be done with a random number generator. i.e.,
- We could assume we have primitive function Random(a,b) which returns an integer between integers a and b inclusive with equally likelihood.
- Algorithms which make use of such a generator are called randomized algorithms.
- In our hiring example we could try to use such a generator to create a random permutation of the input and then run the hiring algorithm on that.

Analysis of the Hiring Problem

- Let Xi be the indicator random variable which is 1 if candidate i is hired and 0 otherwise.
- Let $X = \sum_{i=1}^{n} X_i$
- By our lemma E[Xi] = Pr{candidate i is hired}
- Candidate i will be hired if i is better than each of candidates 1 through i-1.
- As each candidate arrives in random order, any one of the first candidate i is equally likely to be the best candidate so far. So E[Xi] =1/i.

More analysis of hiring problem

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} 1/i = \ln n + O(1)$$

Lemma Assume that the candidates are presented in random order, then algorithmHire-Assistant has a hiring cost of O(chln n)

Proof. From before hiring cost is O(m*ch)where m is the number of candidate hired.From the lemma this is O(ln n).

RANDOMIZED_QUICKSORT

In the randomized version of Quick sort we impose a distribution on input. This does not improve the worst-case running time independent of the input ordering.In this version we choose a random key for the pivot. Assume that procedure Random (*a, b*) returns a random integer in the range [*a, b*); there are *b-a*+1 integers in the range and procedure is equally likely to return one of them. The new partition procedure, simply implemented the swap before actually partitioning.

RANDOMIZED_PARTITION (*A, p, r*)

i ← RANDOM (*p, r*)

Exchange *A*[*p*]↔ *A*[*i*]

return PARTITION (*A, p, r*)

Now randomized quick sort call the above procedure in place of PARTITION

RANDOMIZED_QUICKSORT (*A, p, r*)

If p < r then

q ← RANDOMIZED_PARTITION (*A, p, r*)

RANDOMIZED_QUICKSORT (*A, p, q*)

RANDOMIZED_QUICKSORT (*A, q+1, r*)

Like other randomized algorithms, RANDOMIZED_QUICKSORT has the property that no particular input elicits its worst-case behavior; the behavior of algorithm only depends on the random-number generator. Even intentionally, we cannot produce a bad input for RANDOMIZED_QUICKSORT unless we can predict generator will produce next.

For example, consider below a randomized version of QuickSort.

A Central Pivot is a pivot that divides the array in such a way that one side has at-least 1/4 elements.

```
// Sorts an array arr[low..high]
randQuickSort(arr[], low, high)
1. If low >= high, then EXIT.
2. While pivot 'x' is not a Central Pivot.
  (i)   Choose uniformly at random a number from [low..high].
        Let the randomly picked number number be x.
  (ii)  Count elements in arr[low..high] that are smaller
        than arr[x]. Let this count be sc.
  (iii) Count elements in arr[low..high] that are greater
        than arr[x]. Let this count be gc.
  (iv)  Let n = (high-low+1). If sc >= n/4 and
        gc >= n/4, then x is a central pivot.
3. Partition arr[low..high] around the pivot x.
4. // Recur for smaller elements
   randQuickSort(arr, low, sc-1)
5. // Recur for greater elements
   randQuickSort(arr, high-gc+1, high)
```

The important thing in our analysis is, time taken by step 2 is O(n).

How many times while loop runs before finding a central pivot?

The probability that the randomly chosen element is central pivot is 1/2. Therefore, expected number of times the while loop runs is 2 Thus, the expected time complexity of step 2 is O(n).

What is overall Time Complexity in Worst Case?

In worst case, each partition divides array such that one side has n/4 elements and other side has 3n/4 elements. The worst case height of recursion tree is $\text{Log}_{3/4}$ n which is O(Log n).

```
T(n) < T(n/4) + T(3n/4) + O(n)
```

```
T(n) < 2T(3n/4) + O(n)
Solution of above recurrence is O(n Log n)
```

Note that the above randomized algorithm is not the best way to implement randomized Quick Sort. The idea here is to simplify the analysis as it is simple to analyze.

Typically, randomized Quick Sort is implemented by randomly picking a pivot (no loop). Or by shuffling array elements. Expected worst case time complexity of this algorithm is also O(n Log n)

NP PROBLEMS AND NP COMPLETE

Status of NP Complete problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

What are NP, P, NP-complete and NP-Hard problems?

P is set of problems that can be solved by a deterministic Turing machine in **P**olynomial time.

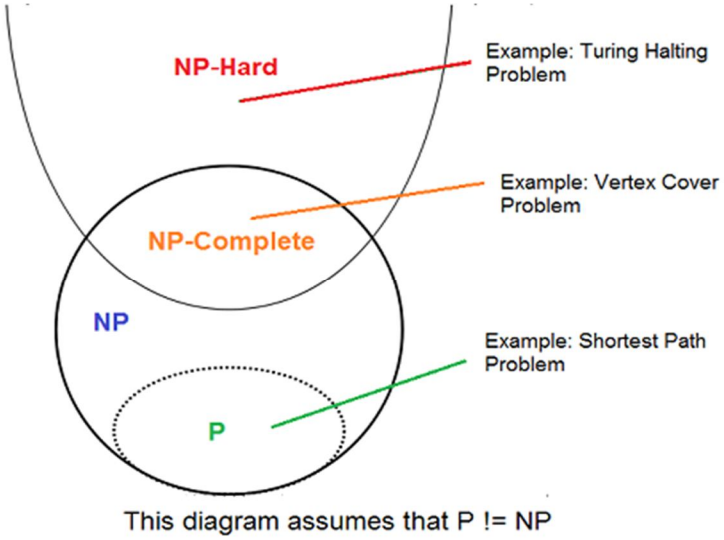
NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).

Informally, NP is set of decision problems which can be solved by a polynomial time via a "Lucky Algorithm", a magical algorithm that always makes a right guess among the given set of choices.

NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:

1. L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
2. Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is NP-Hard if it follows property 2 mentioned above, doesn't need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.



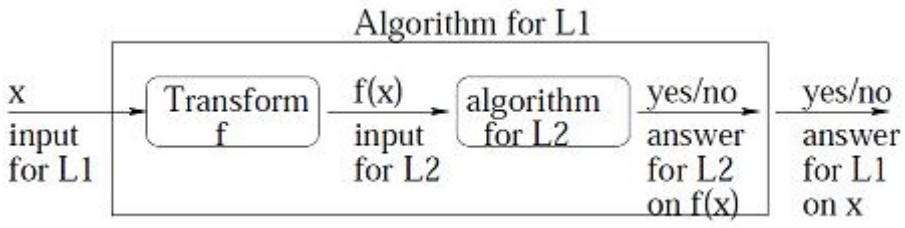
Decision vs Optimization Problems

NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems.

For example, consider the vertex cover problem (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

What is Reduction?

Let L_1 and L_2be two decision problems. Suppose algorithm A_2solves L_2. That is, if y is an input for L_2 then algorithm A_2will answer Yes or No depending upon whether y belongs toL_2 or not. The idea is to find a transformation from L_1 to L_2so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.



Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for

Dijkstra's algorithm to find shortest path, we can take log of all weights and use Dijkstra's algorithm to find the minimum product path rather than writing a fresh code for this new problem.

How to prove that a given problem is NP complete?

From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete. By definition, it requires us to that show every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L. If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

What was the first problem proved as NP-Complete?

There must be some first NP-Complete problem proved by definition of NP-Complete problems. SAT (Boolean satisfiability problem) is the first NP-Complete problem proved by Cook

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don't know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem as NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.